33-rd Annual Simulation Symposium; Washington, D.C., April 16-20, 2000, pp.91-98.

Event–Driven Simulation of Timed Petri Net Models

W.M. Zuberek
Department of Computer Science
Memorial University of Newfoundland
St.John's, Canada A1B 3X5
e-mail: wlodek@cs.mun.ca

Abstract

A collection of software tools for analysis of timed Petri nets, known as TPN–tools*, developed over a number of years, has been extended by a simulation tool,* TPNsim*. The paper gives a brief characterization of* TPNsim*, and discusses some implementation aspects in greater detail.*

1. Introduction

Petri nets have been proposed as a simple and convenient formalism for modeling systems that exhibit parallel and concurrent activities [2, 16, 12]. In order to take also the durations of these activities into account, several types of Petri nets 'with time' have been proposed by assigning 'firing times' to the transitions or 'firing delays' to places of a net. In timed nets, firing times are associated with transitions, and transition firings are 'real–time' events, i.e., tokens are removed from input places at the beginning of the firing period, and they are deposited to the output places at the end of this period (sometimes this is also called a 'three–phase' firing mechanism). In stochastic (and generalized stochastic) Petri nets [10, 3], (exponentially distributed) firing delays (also called 'firing times') are associated with transitions, but the tokens remain (for the duration of the firing delay) in their places, and the instantaneous firings occur at the end of the firing delays. In time nets [8, 1], there is a time interval associated with a transition, and the (instantaneous) firing must occur within this interval of time.

In timed nets, all firings of (enabled) transitions are initiated in the same instants of time in which the transitions become enabled (although some enabled transition cannot initiate their firing; for example, all transitions in a free–choice class can be enabled, but only one can fire). If, during the firing period of a transition, the transition becomes enabled again, a new, independent firing can be initiated, which overlaps with the other firing(s). There is no limit on the number of simultaneous firings of the same transition (sometimes this is called 'infinite firing semantics'). Similarly, if a transition is enabled 'several times' (i.e., it remains enabled after initiating a firing), it may start several independent firings in the same time instant.

In timed nets, the firing times of some transitions may be equal to zero, which means that the firings are instantaneous; all such transitions are called immediate (while the other are called timed). Since the immediate transitions have no tangible effect on the (timed) behavior of the model, it is convenient to split the set of transitions into two parts, the set of immediate and the set of timed transitions, and to fire first all (enabled) immediate transitions; only when no more immediate transitions are enabled, the firings of (enabled) timed transitions are initiated (still in the same instant of time). It should be noted that such a convention effectively introduces the priority of immediate transitions over the timed ones, so the conflicts of immediate and timed transitions should be avoided. Also, the free–choice classes of transitions must be 'uniform', i.e., all transitions in each free–choice class must be either immediate or timed.

The firing times of transitions can be either deterministic or stochastic (i.e., described by a probability distribution function); in the first case, the corresponding timed nets are referred to as D–timed, in the second, for the (negative) exponential distribution of firing times, the nets are referred to as M–timed (or Markovian) nets. In both cases, the concepts of states and state transitions have been formally defined and used in derivation of different performance characteristics of the models [17].

Timed Petri net models are discrete–state systems; the states of net models change, as the result of transition firings, by removing or adding tokens to places (for both instantaneous and timed firings). Analysis of timed models by using discrete–event simulation is thus a 'natural' approach to model evaluation, which imposes very few restrictions on the class of analyzed models (other approaches, like reachability analysis and structural analysis, can be applied only to certain classes of net models).

In discrete–event simulation, an occurrence of an event may cause some other events to happen at the same time or at some future time. There are two basic methods of organizing discrete–event simulations [11], the time–based (or synchronous–timing [7] or fixed–step [4]) simulation and event–driven (or asynchronous–timing [7]) simulation. For time–based simulation, the model is analyzed at consecutive, uniformly distributed time instants (the time–step is constant), and all events which can occur at these time instants, are executed (changing the state of the model); although this approach is rather simple to implement, quite often it also is very inefficient, especially when events are clustered in time (i.e., when there are periods of high activities followed by periods of inactivities of the model).

In event–driven simulation, the control of the (simulated) time depends only upon the activities of the model. During execution of events, all future events are stored in a list of events also called the 'event queue'. This list is ordered with respect to the time in which the events are scheduled to occur; the event scheduled to occur in the nearest future is at the front of the list. If there are no more events to be executed at the present (simulated) time instant, the 'first' event is fetched (or 'dequeued') from the list of events, the (simulated) time is advanced accordingly, and the event is executed (possibly creating new events). Event–driven simulation introduces a small overhead in comparison to the time–based simulation, but it is much more flexible with respect to time control, as it analyzes the model only at time instants when events occur.

The results of simulation runs may include different performance measures, evaluated and presented in many different ways. Quite often these results may depend upon a particular application. Therefore, in *TPNsim*, only very simple data are collected during the simulation of net models, and some 'postprocessing' capabilities are provided to calculate performance measures from these 'raw' simulation results.

2. Description of net models

In *TPN–tools*, net descriptions are 'transition oriented', i.e., nets are specified as collections of transitions, and each transition contains all parameters associated with it.

The (simplified) syntax of model descriptions, in the BNF notation, is as follows [18]:

```
<model-descr> ::= <net-descr> <imarking>
<net-descr> ::= <net-header> ( <transition-list> )
<net-header> ::= Mnet | Dnet | net
<transition-list> ::= <transition> |
            <transition-list> ; <transition>
<transition> ::= <t-header> = <input-output-list>
<t-header> ::= <t-indent> <type> <time> <prob>
<t-ident> ::= <integer> | <name>
<type> ::= :D | :M | <empty>
<time> ::= * <rational> | <empty>
<prob> ::= , <rational> | , [ <place-id> ] |
             , <integer> / <integer>
<rational> ::= <integer> | <integer> . <integer>
<input-output-list> ::= <input-list> |
            <input-list> / <output-list>
<input-list> ::= <arc> | <input-list> , <arc>
<output-list> ::= <arc> | <output-list> , <arc>
<arc> ::= <place-id> | <place-id> : <weight>
<place-id> ::= <integer> | <name>
<weight> ::= <integer>
<name> ::= <letter> | <name> <letter> |
            <name> <digit>
<imarking> ::= mark ( <marking-list> )
<marking-list> ::= <marked-place> |
            <marking-list> , <marked-place>
<marked-place> ::= <place> | <place> : <count>
<count> ::= <integer>
```

The type on the net (D–timed or M–timed) is indicated in the net header. The type of a transition (M–timed, D–timed) can also be indicated by the `type` elements; such a specification overrides the net type.

Transitions with empty `time` elements denote immediate transitions, i.e., transitions with the firing time equal to 0.

Probability element `prob` specifies the free-choice probabilities of transitions or relative frequencies of conflicting transitions. Empty element `prob` is equivalent to probability equal to 1. Dynamic (i.e., marking–dependent) relative frequencies are indicated by place references of the `prob` element. During conflict resolution, the number of tokens in the indicated place is used as the relative frequency of transition firings. Usually this reference place is one of the transition's input places. The form `<integer>/<integer>` is provided as a convenient way of specifying fractional values.

Arcs without weight are equivalent to arcs with weight equal to 1. Inhibitor arcs are specified as arcs with weight equal to 0.

Marked places without the `count` element are equivalent to places with the value of `count` equal to 1.

3. Simulation of net models

The simulation of net models is based on the event queue [14, 9]. All future events are stored in the event queue in an increasing order of their occurrence time.

The simulation is usually executed for some fixed period of (simulated) time; let this period of time be

denoted by `TimeLimit`. This time limit can be incorporated in the simulator by scheduling a special event, `EndSimulation`, at time `TimeLimit`; execution of this event terminates the simulation and outputs simulation results.

If `schedule(Event,Time)` denotes the operation of inserting a new `Event` that is to occur at time `Time`, a general outline of event–driven simulation can be as follows:

```
schedule(EndSimulation,TimeLimit);
schedule(InitialEvent,0.0);
while nonempty(EventQueue) do
   dequeue(EventQueue,Event);
   SimulatedTime := Event.Time;
   execute(Event)
endwhile;
```

It should be observed that if no new events are created during execution of the initial event (or subsequent events), the simulation run ends rather quickly.

In Petri net models, the events correspond to the terminations of transitions' firings (and possibly starting new firings). The execution of an event (`execute(Event)`) is composed of three consecutive steps: (i) all firings which are scheduled to terminate at the current event are processed and the net marking is updated by depositing tokens to the corresponding output places of transitions; (ii) all immediate transitions are fired iteratively until no immediate transitions are enabled (the marking function is updated during all these firings); (iii) the new firings of timed transitions are initiated by scheduling their termination according to their (deterministic or stochastic) firing times. All conflicts which may need to be resolved during execution of an event use random numbers to make the required choices (for example, in free–choice structures) and to sample the corresponding probability distributions (e.g., to determine the actual firing times).

Simulation of the behavior of net models is performed by the directive:

```
simulate(tlimit);
```

where `tlimit` is the simulation time limit (in the same units as the firing times of transitions). During the simulation, the numbers of firings are counted for each transition, and also the total time of all firings (including overlapping firings) is cumulated for each transition. For D–timed firings, the firing times are constant, as specified in the net description; for M–timed firings, the firing times are generated by a random number generator (with exponential distribution), using the average firing time as a (scaling) parameter. Moreover, the numbers of tokens entering each place are also counted (including the initial marking), and the total 'waiting' time of all tokens is cumulated for each place. After the termination of simulation, these 'raw data' can be displayed by the directive:

```
simres;
```

which outputs a list of firing counts and total firing times for all 'active' transitions (i.e., transitions with nonzero firing counts); for immediate transitions, only firing counts are provided. Similarly, the token counts are output for all 'active' places together with the total (nonzero) waiting times.

Example. The net shown in Fig.1 is a simple illustration of 'marking–dependent' conflict resolutions [18]. The net represents an interactive system executing two classes of jobs, say class–A and class–B jobs, with random selection of jobs from a common pool of waiting jobs. As no priorities and no queueing is assumed, the probability of selection of a class–A job (if there is any such job waiting) is determined by the ratio of the number of waiting class–A jobs to the total number of waiting jobs.

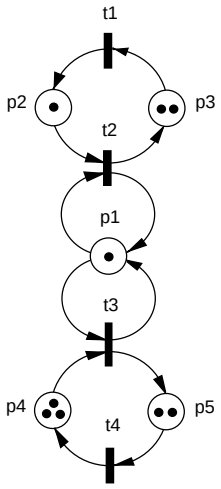

Fig.1. A processor executing two classes of jobs.

In the model, p_1 represents the available (or idle) processor. Execution of class–A jobs is represented by t_2, and class-B jobs – by t_3. t_1 models the 'thinking time' for class–A jobs, and t_4 – the same for class–B jobs. p_2 is the pool of waiting class–A jobs, p_4 – the pool of waiting class–B jobs. t_2 and t_3 are in conflict because of sharing p_1, and the relative frequencies of t_2 and t_3 firings can be determined by the token counts in p_2 and p_4, respectively.

Assuming that all firing times are exponentially distributed, the description of the model and its simulation results can be as follows:

```
Mnet(t1*5=p3/p2;
     t2*3,[2]=p1,p2/p1,p3;
     t3*2,[4]=p1,p4/p1,p5;
     t4*8=p5/p4)
mark(p1,p2,p3:2,p4:3,p5:2);
simulate(10000);
simres;

Simulation Counts:
 firings of t1 : 1707 total firing time 8517.73
 firings of t2 : 1705 total firing time 5021.65
 firings of t3 : 2392 total firing time 4793.9
 firings of t4 : 2393 total firing time 18836.8

 tokens in p1 : 4097 total waiting time 44.311
 tokens in p2 : 1705 total waiting time 16411.8
 tokens in p3 : 1707
 tokens in p4 : 2393 total waiting time 26298.5
 tokens in p5 : 2393
```

To check the effect of marking–dependent frequencies, these simulation results can be compared with the case when the relative frequencies of (conflicting) firings are constant and are the same for both transitions (which is the default case):

```
Mnet(t1*5=p3/p2;
     t2*3=p1,p2/p1,p3;
     t3*2=p1,p4/p1,p5;
     t4*8=p5/p4)
mark(p1,p2,p3:2,p4:3,p5:2);
simulate(10000);
simres;

Simulation Counts:
 firings of t1 : 1819 total firing time 9436.65
 firings of t2 : 1818 total firing time 5487.67
 firings of t3 : 2197 total firing time 4335.24
 firings of t4 : 2199 total firing time 17034.1

 tokens in p1 : 4015 total waiting time 37.7378
 tokens in p2 : 1819 total waiting time 15041.7
 tokens in p3 : 1819
 tokens in p4 : 2202 total waiting time 28512
 tokens in p5 : 2199
```

Intuitively, class–B jobs should be disadvantaged in this case because, on average, more waiting class–B jobs (in p_4) are expected than class–A jobs (in p_2), so with the same probabilities of selection, class–B jobs will be selected less often than in the first case (i.e., when marking–dependent frequencies are used). The results confirm this observation.

4. Postprocessing

The ('raw') simulation results are stored in the internal structure representing the net's elements [18], so after a simulation run, the results of simulation can be used for evaluation of different performance measures. A 'postprocessing' interpreter is provided which evaluates simple expressions composed of constants, net parameters and simulation results. The interpreter is invoked by the command `comp` with an expression as its argument:

```
<command> ::= comp ( <expr> ) ;
<expr> ::= <term> | <expr> + <term |
           <expr> - <term>
<term> ::= <fact> | <term> * <fact |
           <term> / <fact>
<fact> ::= <number> | <t-arg> | <p-arg> |
           <s-name> | ( <expr> )
<s-name> ::= STime | SimTime
<t-arg> ::= t ( <t-ident> ) | n ( <t-ident> ) |
            f ( <t-ident> ) | p ( <t-ident> )
<p-arg> ::= t ( <p-ident> ) | n ( <p-ident> ) |
            m ( <p-ident> )
<t-ident> ::= <integer> | <name>
<p-ident> ::= <integer> | <name>
```

The 'special name' (`s-name`) `STime` (or `SimTime`) denotes the simulation time (`tlimit` of the last `simulate` directive).

The `t(...)` arguments refer to the total (cumulative) firing times (of transitions) or total token waiting times (in places). The `n(...)` arguments refer to the numbers of firings (of transitions) or the numbers of tokens entering the places (including the initial marking). The `f(...)` arguments refer to the firing times of transitions, and the `p(...)` arguments, to the choice probabilities or relative frequencies assigned to transitions.

Example. Some simple values calculted from the previous results (Section 2) are as follows:

```
comp(t(t2)/n(t2));  (* throughput A *)
   value : 2.94525
comp(t(t3)/n(t3));  (* throughput B *)
   value : 2.00414
comp(t(p2)/n(p2));  (* average waiting time A *)
   value : 9.62567
comp(t(p4)/n(p4));  (* average waiting time B *)
   value : 10.98977
comp((t(t2)+t(t3))/STime);   (* proc. util. *)
   value : 0.98155
```

Because in some cases the formulas become quite complicated, a mechanism for declaring variables and evaluating subexpressions is also provided:

```
<command> ::= <declaration> | <assignment>
          | comp ( <expr> )
<declaration> ::= var <list-of-variables> ;
```

```
<list-of-variables> ::= <name>
          | <list-of-variables> , <name>
<assignment> ::= <variable> := <expr> ;
<variable> ::= <name>
```

Variables (after assigning values to them) can be used in expressions in the same way as the 'special names'. All variables must be declared before the first use, and each declaration of variables erases all previous declarations.

5. Implementation issues

The 'event queue' is implemented as a two–level list structure, with the first level representing the (future) time instants (in ascending order), and the second level, for each time instant, represents all firings which are scheduled to terminate at this time instant. The organization of the event queue uses its own (dynamic) memory management in the sense that it maintains a list of 'released' descriptors, and when a new descriptor is needed, a check is first made if a released descriptor is available; only when all descriptors are used, a new descriptor is created from additionally allocated memory. This solution usually results in a very efficient usage of memory as typically only a few event descriptors are allocated and they are frequently reused during the simulation.

Processing for each time instant is done in three consecutive steps. First, all firings which are scheduled for termination as the current event are processed and the marking function is updated by depositing tokens to transitions' output places (the simulation statistics for places are also updated at the same time).

In the second step, the firings of enabled immediate transitions are executed iteratively (and the marking function is updated accordingly) until the set of enabled immediate transitions is empty. The enabled (immediate) transitions are fired in three groups; first, all (enabled) conflict–free transitions are fired (in any order because they are independent one from another); then, for each free-choice class of enabled transitions, one transition is selected in a random way and fired, while all remaining transitions in this class are disabled; finally, the conflict classes are processed one after another, also by randomly selecting a transition, firing it and disabling all other transitions in this conflict class:

```
Enabled := bag_of_enabled_immediate_transitions;
while nonempty(Enabled) do
   fire_all_enabled_conflict_free_transitions;
   for each enabled_free_choice_class do
      randomly_select_a_transition_in_this_class;
      fire_the_selected_transition;
      disable_other_transitions_in_this_class
   endfor;
   for each enabled_conflict_class do
      randomly_select_a_transition_in_this_class;
      fire_the_selected_transition;
      disable_other_transitions_in_this_class
   endfor;
   Enabled := bag_of_enabled_immediate_transitions
endwhile;
```

(the collection of enabled transitions is defined as a bag [13] rather than a set to allow multiple occurrences of transitions which are enabled 'several times').

The third step initializes the firings of enabled timed transitions (again updating the marking function for each initialized firing):

```
Enabled := bag_of_enabled_timed_transitions;
for each enabled_conflict_free_transition do
   initiate_transition_firing;
   determine_the_firing_time(FTime);
   schedule(transition,SimulatedTime+FTime)
endfor;
for each enabled_free_choice_class do
   randomly_select_a_transition_in_this_class;
   initiate_transition_firing;
   determine_the_firing_time(FTime);
   schedule(transition,SimulatedTime+FTime);
   disable_other_transitions_in_this_class
endfor;
while nonempty(Enabled) do
   for each enabled_conflict_class do
      randomly_select_a_transition_in_this_class;
      initiate_transition_firing;
      determine_the_firing_time(FTime);
      schedule(transition,SimulatedTime+FTime);
      disable_other_transitions_in_this_class
   endfor;
   Enabled := bag_of_enabled_timed_transitions
endwhile;
```

The length of the firing time is determined depending upon the type of transition or the type of net. For D–timed firings, `FTime` is simply the firing time of the transition; for M–timed firings, an exponentially distributed random number generator is used with the transition's average firing time as a parameter.

Identification of (enabled) conflict–free transitions and (enabled) free–choice classes of transitions is done during the initial processing of the net; all conflict–free transitions have the `class` field in the descriptor [18] set to `1`, the same field for free–choice classes of transitions is set to a value greater than `1` (the same value for all transitions in the same free–choice class), and for all other transitions (potentially in conflicts) `class` is set to `0`.

For free–choice classes of transitions (assumed to be linked into a list `Class`), and for a conflict class

(also linked into a list `Class`), the selection of a transition for firing (`randomly_select_a_transition`) uses a uniformly distributed random number generator `UniformRandomNumber`:

```
prob := UniformRandomNumber;
select := NIL;
while select = NIL and nonempty(Class) do
   if prob < Class.prob then select := Class
   else
      prob := prob - Class.prob;
      next(Class)
   endif
endwhile;
if select = NIL then
   error("unsuccessful selection.-");
```

Since the classes of conflicting transitions are marking–dependent, they must be determined for each marking function. Assuming that all enabled transitions (without conflict–free and free–choice transitions) are linked in a list `List`, and that the conflict class will be represented by another linked list, `Class` (which is empty initially), the determination of a conflicting class is as follows. The first transition from `List` is moved to `Class`, and then, iteratively, those transitions from `List` which share any places with transitions in `Class` are moved from `List` to `Class`:

```
Class := head(List);
List := tail(List);
cont := true;
while cont do
   cont := false;
   for each TransL in List do
      for each TransC in Class do
         if nonempty(shared(TransL.input,
         TransC.input)) then
            delete(TransL,List);
            insert(TransL,Class);
            cont := true
         endif
      endfor
   endfor
endwhile;
```

where `shared(list1,list2)` determines the set of shared places on the lists `list1` and `list2` (the input lists of transitions `TransL` and `TransC` above).

The process of identifying conflict classes for a given marking function, and in fact, for a given list of enabled transitions, is rather time–consuming. Therefore, instead of repeatedly checking possible conflicts by comparing the input lists of transitions, it may be beneficial to save the performed decompositions of the original lists `List` into `Class` and the remaining list `List` (which may be empty), and to reuse these saved decomposition if the same (original) list `List` needs to be analyzed again.

The decompositions can be (optionally) saved in a (dynamically created) tree–like structure in which consecutive layers of the tree are selected by consecutive transitions of the (ordered) original list of (enabled) transitions, and in which the nodes are associated with pairs of decomposed list. There is a limit on the (total) number of nodes in the decomposition tree. If this limit is exceeded, the saving option is automatically turned off and the (partial) decomposition tree is deleted. It should be noted that this option is justified only when the set of possible marking functions is rather small, so there is a good chance of reusing the decompositions. If this is not the case, the use of this option can actually increase the simulation time as many new decompositions need to be stored, consuming large amounts of memory and performing many useless searches of the decomposition tree.

Example. Fig.2 shows a model of "five dining philosophers", in which places F1, ..., F5 represent the forks, and each of the five philosophers is modeled by a cyclic subnet with transition Xth representing a thinking philosopher X (X = A, ..., E), transition Xet – an eating philosopher X, place Xrd – a philosopher ready to eat, and place Xfd – a philosopher who finished eating, returned the two forks, and is going to think.

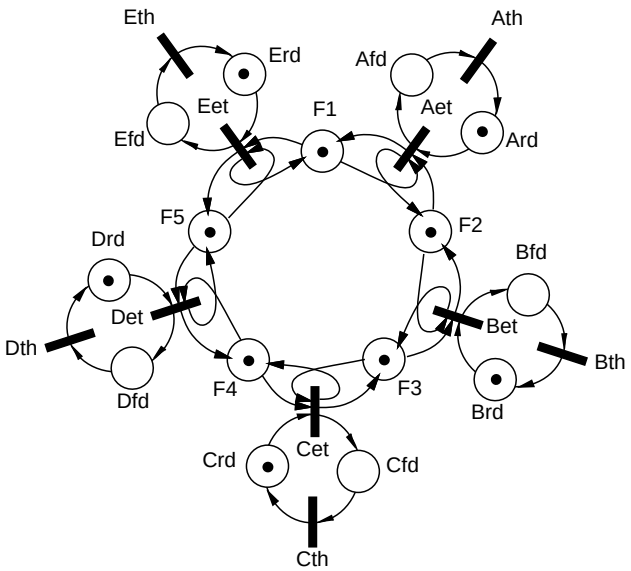

Fig.2. Five dining philosophers.

In Fig.2, transitions Ath, ..., Eth are conflict–free transitions, while the remaining transitions Aet, ..., Eet constitute one conflict class (because of shared places

F1, ..., F5); Fig.2 shows that all these transitions can be enabled at the same time. However, only two out of these five transitions can fire simultaneously, and there are 10 different combinations of firing transitions. All these possibilities correspond to different resolutions of the conflicts. For example, if Aet is selected for firing, Bet and Eet will be disabled (because the firing of Aet removes the tokens from F1 and F2), so only Cet or Det can fire simultaneously with Aet. Similarly, Bet can fire simultaneously only with Det or Fet, and so on.

The concept of decompositions can be illustrated using a different marking, for example {Ard,Brd,Drd,F1,F2,F3,F4,F5} (there is no decomposition for the marking shown in Fig.2; all five transitions are in a single conflict class). For this new marking, the set of enabled transitions {Aet,Bet,Det} is decomposed into two classes, {Aet,Bet} and {Det}, as the input sets of Aet and Bet share place F2, but there is no element shared by Det and Aet or Bet. The first of these two classes is a conflict class, while the second is a conflict–free class (because it contains just one transition). In the decomposition tree, the path `Aet.Bet.Det` will be associated with an entry `[Aet,Bet + Det]`, describing this decomposition.

The decomposition tree for the model shown in Fig.2 is as follows (the listing shows the levels of the structure and the actual decompositions enclosed in brackets '[' and ']'; the element '`NIL`' in these decompositions denotes the empty class). A node corresponding to the set of enabled transitions is obtained by tracing the path from the root (consecutive levels are indicated by parenthesized sublists, so the path Aet.Bet.Det is identified as "Aet(Bet(...,Det[...]...") using the enabled transitions in alphabetical order:

```
Aet(Bet[Aet,Bet + NIL]
       (Cet[Aet,Bet,Cet + NIL]
           (Det[Aet,Bet,Cet,Det + NIL]
               (Eet[Aet,Bet,Cet,Det,Eet + NIL]),
            Eet[Aet,Bet,Cet,Eet + NIL]),
        Det[Aet,Bet + Det]
           (Eet[Aet,Bet,Eet,Det + NIL]),
        Eet[Aet,Bet,Eet + NIL]),
    Cet[Aet + Cet]
       (Det[Aet + Cet,Det]
           (Eet[Aet,Eet,Det,Cet + NIL]),
        Eet[Aet,Eet + Cet]),
    Det[Aet + Det]
       (Eet[Aet,Eet,Det + NIL]),
    Eet[Aet,Eet + NIL]),
Bet(Cet[Bet,Cet + NIL]
       (Det[Bet,Cet,Det + NIL]
           (Eet[Bet,Cet,Det,Eet + NIL]),
        Eet[Bet,Cet + Eet]),
    Det[Bet + Det]
       (Eet[Bet + Det,Eet]),
    Eet[Bet + Eet]),
Cet(Det[Cet,Det + NIL]
       (Eet[Cet,Det,Eet + NIL]),
    Eet[Cet + Eet]),
Det(Eet[Det,Eet + NIL])
```

The decomposition tree is also sketched in Fig.3 where nodes without any decomposition are shown by 'white' circles, and the nodes associated with decompositions – by dark circles. The set of enabled transitions to be verified for decomposition is represented by a path from the root (in alphabetical order of transitions); for example, the set {Aet,Cet,Det} is decomposed into (see the list above) a single–transition class {Aet} and the remaining set {Cet,Det} which cannot be decomposed further.

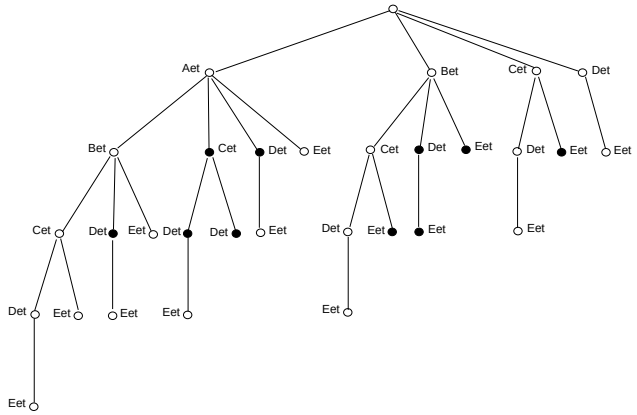

Fig.3. The decomposition tree for the net in Fig.2.

There are 10 effective decompositions and 30 nodes in this decomposition tree. In a typical simulation run, the transitions are fired many thousands times, so without saving the decompositions, the same conflicts would be (unnecessarily) analyzed many thousands of times.

6. Concluding remarks

An event–driven simulation tool, recently added to *TPN–tools*, provides a simple and flexible means for characterizing the behavior of timed models. The tool can be especially useful for evaluations of models for which other methods (for example, reachability analysis) cannot be used because of the size of the state space or unsatisfied constraints (for example, unboundedness of models).

At present, there are only two types of temporal information that can be associated with transitions: constant firing times and exponentially distributed firing times. Several other distributions can easily be implemented by minor entensions to the model description

language [18]. For example, in addition to `D` and `M`, some other transition *types* can be defined, possibly including a 'user–defined' type (with some sort of standardized interface to allow a functional description of the required distribution). Similarly, more elaborate forms of simulation results can be adopted from other simulation languages and/or tools.

At present, *TPNsim* cannot handle high–level nets, and only a very restricted form of colored nets [6] can be used. Improved capabilities for high–level net models need to be incorporated into *TPNsim* in order to simplify analysis of large models.

Another useful extension of *TPNsim* would be to support event–driven as well as time–based simulation; it appears that the simulation of instruction–level architectures is often performed at the processor cycle level, in which case the time–based simulation can be more efficient than the event–driven one.

The discussion of the implementation issues focuses on one aspect of *TPNsim*, its processing of conflicts. There are two basic considerations behind this decision. One is that processing of conflicts is one of the most interesting aspects of the simulation of net models; many ather aspects are quite similar to other discrete–event formalisms. The second consideration is that although there are many other modeling tools based on Petri nets [20], processing of conflicts is either restricted to a few special cases (e.g., free–choice nets), or described through a number of 'standard' examples, which are of little help when a 'non–standard' case needs to be analyzed.

The described simulation tool was used extensively in several evaluation studies, including a simulation of a multilayered model of communication networks [15] and multithreaded distributed memory multiprocessor systems [5, 19]. *TPNsim*'s performance and flexibility compared quite favorably with some other Petri net tools; for example, it was an order of magnitude faster and much more flexible than Visual SimNet [15].

References

[1] van der Aalst, W.M.P., "Interval timed colored Petri nets and their analysis"; in: "Applications and Theory of Petri Nets 1993" (Lecture Notes in Computer Science 691); Ajmone Marsan, M. (ed.), pp.453–472, Springer–Verlag 1993.

[2] Agerwala, T., "Putting Petri nets to work"; IEEE Computer Magazine, vol.12, no.12, pp.85-94, 1979.

[3] Ajmone Marsan, M., Conte, G., Balbo, G., "A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems"; ACM Trans. on Computer Systems, vol.2, no.2, pp.93–122, 1984.

[4] Ferrari, D., "Computer systems performance evaluation"; Prentice–Hall 1978.

[5] Govindarajan, R., Suciu, F., Zuberek, W.M., "Timed Petri net models of multithreaded multiprocessor architectures"; Proc. 7-th Int. Workshop on Petri Nets and Performance Models (PNPM'97), St. Malo, France, pp.153-162, 1997.

[6] Jensen, K., "Coloured Petri nets"; in: "Advanced Course on Petri Nets 1986" (Lecture Notes in Computer Science 254), Rozenberg, G. (ed.), pp.248-299, Springer–Verlag 1987.

[7] Kobayashi, H., "Modeling and analysis – an introduction to system performance evaluation methodology"; Addison–Wesley 1981.

[8] Merlin, P.M., Farber, D.J., "Recoverability of communication protocols – implications of a theoretical study"; IEEE Trans. on Communications, vol.24, no.9, pp.1036–1049, 1976.

[9] Mitrani, I., "Simulation techniques for discrete event systems"; Cambridge University Press 1982.

[10] Molloy, M.K., "Performance analysis using stochastic Petri nets"; IEEE Trans. on Computers, vol.31, no.9, pp.913–917, 1982.

[11] Molloy, M.K., "Fundamentals of performance modeling"; Macmillan 1989.

[12] Murata, T., "Petri nets: properties, analysis and applications"; Proceedings of IEEE, vol.77, no.4, pp.541–580, 1989.

[13] Peterson, J.L., "Petri net theory and the modeling of systems"; Prentice-Hall 1981.

[14] Pooch, U., "Discrete–event simulation — a practical approach"; CRC Press 1993.

[15] Reid, M., "Modeling and performance analysis of ATM LANs"; M.Sc. Thesis, Department of Computer Science, Memorial University of Newfoundland, St. John's, Canada A1B 3X5, 1996.

[16] Reisig, W., "Petri nets - an introduction" (EATCS Monographs on Theoretical Computer Science 4); Springer–Verlag 1985.

[17] Zuberek, W.M., "Timed Petri nets – definitions, properties and applications"; Microelectronics and Reliability (Special Issue on Petri Nets and Related Graph Models), vol.31, no.4, pp.627–644, 1991.

[18] Zuberek, W.M., "Modeling using timed Petri nets – model description and representation"; Technical Report #9601, Department of Computer Science, Memorial University of Newfoundland, St. John's, Canada A1B 3X5, 1996 (available through anonymous `ftp` at `ftp.cs.mun.ca` as `/pub/techreports/tr-9601.ps.Z`).

[19] Zuberek, W.M., Govindarajan, R., "Performance balancing in multithreaded multiprocessor architectures"; Proc. 4-th Australasian Conf. on Parallel and Real–Time Systems (PART'97), Newcastle, Australia, pp.15-26, 1997.

[20] DAIMI (Department of Computer Science, University of Aarhus, Denmark) maintains a database of Petri net tools, `http://www.daimi.au.dk/PetriNets/tools`.